John von Neumann Institute for Computing

OpenMP Parallelizations of Viswanathan and Bagchi's Algorithm for the Two Dimensional Cutting Stock Problem

G. Miranda Valladares, C. León Hernández

published in

Parallel Computing:
Current & Future Issues of High-End Computing,
Proceedings of the International Conference ParCo 2005,
G.R. Joubert, W.E. Nagel, F.J. Peters, O. Plata, P. Tirado, E. Zapata (Editors),
John von Neumann Institute for Computing, Jülich,
NIC Series, Vol. **33**, ISBN 3-00-017352-8, pp. 285-292, 2006.

http://www.fz-juelich.de/nic-series/volume33

OpenMP Parallelizations of Viswanathan and Bagchi's Algorithm for the Two Dimensional Cutting Stock Problem *

Gara Miranda Valladares[a], Coromoto León Hernández[a]

[a] Departamento de Estadística, I. O. y Computación,
University of La Laguna, Tenerife E-38271, Spain.
(gmiranda | cleon)@ull.es

1. Introduction

This paper presents two proposals of parallelization of Viswanathan and Bagchi's algorithm to solve the Two-Dimensional Cutting Stock Problem [12]. Both implementations use the skeleton library MaLLBa [1]. This library provides algorithmic skeletons for solving combinatorial optimization problems. Viswanathan and Bagchi's algorithm is based on a best-first search, so it is easily adaptable to different MaLLBa skeleton interfaces. More precisely, to solve the problem the MaLLBa::BnB [6] and MaLLBa::A* [9] skeletons have been instantiated. These skeletons require from the user the implementation of several C++ classes: a `Problem` class defining the problem data structures, a `Solution` class to represent the result and a `SubProblem` class to specify subproblems. Initially, the problem was implemented using the MaLLBa::BnB interface. Due to the dependent generation of subproblems done by the problem algorithm, the distributed parallel solver provided by this interface was not able to afford the resolution of the problem. An ad hoc parallelization was done over the MaLLBa::BnB sequential solver. Then, the problem was also implemented with the MaLLBa::A* interface. This interface provides a sequential and a parallel solver for A* searches. In this case, the interface has the hoped behaviour when the generation of new subproblems depends on other subproblems previously generated. Both parallel implementations use the shared memory paradigm and the OpenMP [10] tool.

The article content will be organized in the following way: First we will introduce the problem and present Viswanathan and Bagchi's algorithm for the problem resolution. Section 3 is dedicated to the description of MaLLBa skeleton classes that implement the problem. The two proposals of parallelization of this problem implementation will be described in detail in section 4. Computational results on a multiprocessor will be shown in section 5. Finally, conclusions and future works are given.

2. Two-Dimensional Cutting Stock Problem

The Two-Dimensional Cutting Stock Problem has lots of applications in many different types of industries. Its formulation is as follows. Consider a surface S with size $L \times W$ made of a certain material. And a set of n different patterns, each one with dimensions $l_i \times w_i$, with an associated profit c_i. Let's b_i the number of available pieces of type i and x_i the number of pieces of type i that have been used. The problem consists in finding the set of patterns and its distribution along the surface S that get a maximum profit and a minimum loss of the material, that is,

*This work has been supported by the EC (FEDER) and by the Spanish Ministry of Science and Technology with contract number TIC2002-04498-C05-05. Also by the Canary Government Project COF2003/022. The work of G. Miranda has been developed under the grant FPU-AP2004-2290.

$$Maximize \sum_{i=1}^{n} c_i x_i \;\; subject\ to \{R\}$$

where R is a set of specific constraints. Depending on the definition of this constraint set, you will have a certain type of Cutting Stock Problem. Anyway, even supposing the simplest constraint set, Cutting Stock Problems are classified as NP-Hard problems [3].

The first formulation of the Cutting and Packing Problem as a Linear Programming Problem was made in 1961 [5]. Since that moment a lot of bibliography about the different definitions of the problem has been appeared. There are many classifications of these problems depending on the number of dimensions, the number of available surfaces and patterns, the shape of the patterns, the orientation, the availability, etc., [4,11]. The solution to the problem has been studied following multiple approximations. Dynamic Programming techniques, Integer Programming methods, Heuristic searches, etc., have been used, [2,3,8]. In 1989, Viswanathan and Bagchi [12] propose an exact algorithm based on a *best-first* search. In [7] Hifi introduces a modification in the calculation of the algorithm bounds.

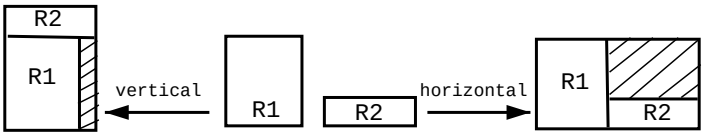

Figure 1. Vertical and horizontal builds

Viswanathan and Bagchi's algorithm considers that any solution to the problem can be obtained by vertical and horizontal combinations of different *builds* of pieces, see Figure 1.

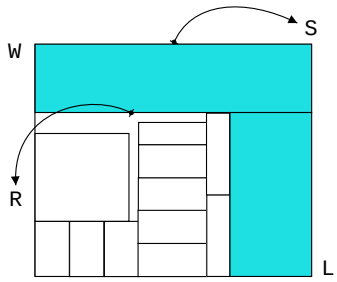

Figure 2. Surface S and build R

At each step, the considered *best-build* is placed in the left bottom corner of the available surface and it is combined with the best builds selected until that moment, see Figure 2.

In order to have a way to know which builds are better and which are worse, it is defined the R *build accumulated profit*, $g(R)$, as the profit sum of all patterns belonging to the build R. Besides, $h(R)$ is defined as the *maximum profit* obtainable from the remaining area of the surface. So, having a certain build R, its *total profit* is defined as: $f(R) = g(R) + h(R)$.

To calculate $f(R)$, the algorithm uses an upper estimation of $h(R)$, denoted as $h'(R)$. To calculate this estimation, functions $U_1(x_R, y_R)$ or $U_2(x_R, y_R)$ are defined:

$$U_1(x_R, y_R) = F(L, W - y_R) + F(L - x_R, W)$$
$$U_2(x_R, y_R) = max\{h_1(x, y), h_2(x, y)\}\ y\ \ U_2(L, W) = 0$$

where F is the function for the Bidimensional Knapsack Problem [5] that is satisfied for all x and y ($x \leq L$ and $y \leq W$) and where $h1$, $h2$ are defined as:

```
begin
  Open := {r_1, r_2, ..., r_n};
  Best := {};
  finished := false;
  repeat
    choose the R rectangle with higher f' value;
    if h'(R) = 0 then
      finished := true;
    else
    begin
      transfer R from Open to Best;
      build all Q guillotine rectangle such as:
        i. Q is an horizontal or vertical build of R with any R' rectangle (included R') from Best;
        ii. Q dimensions are ≤ (L, W);
        iii. Q satisfies all the problem constrains
      put all new Q rectangles in Open with their appropiate g, h' and f' values.
    end;
  until finished;
  Return R as the problem solution;
end
```

Figure 3. Viswanathan and Bagchi's Algorithm

$$F(x,y) = max\{F_0(x,y), F(x_1,y) + F(x_2,y), F(x,y_1) + F(x,y_2)\}$$
$$F_0(x,y) = max\{\{0,c_i\} : l_i \leq x, w_i \leq y, \forall\, i \in 1,\ldots,n\}$$
$$x \geq x_1 + x_2, \quad 1 \leq x_1 \leq x_2$$
$$y \geq y_1 + y_2, \quad 1 \leq y_1 \leq y_2$$
$$h_1(x,y) = max\{U_2(x+u,y) + F(u,y) : 1 \leq u \leq L - x\}$$
$$h_2(x,y) = max\{U_2(x,y+v) + F(x,v) : 1 \leq v \leq W - y\}$$

So, having a certain build R, its *estimated total profit*, $f'(R)$, is defined as: $f'(R) = g(R) + h'(R)$.

Figure 3 shows a pseudocode of *Viswanathan and Bagchi's algorithm*. The presented method follows a scheme very similar to an A* search, using the "total profit" and "estimated total profit" functions described above.

3. The MaLLBa Library

An *Algorithmic Skeleton* must be understood as a set of procedures that compose the structure to use in the development of programs for the resolution of a given problem using a particular algorithmic technique. They provide an important advantage in comparison to a direct implementation of the algorithm from the beginning, not only in terms of code reuse but also in methodology and concept clarity. Skeletons introduce modularity in the design of algorithms. In general, the software that supply skeletons presents declarations of empty classes. The user must fill these empty classes to adapt the given scheme for the resolution of a particular problem.

The problem has been implemented using MaLLBa::BnB and MaLLBa::A* skeletons. User interfaces for both schemes are very similar. The user has to specify similar methods and classes. The main difference between the two skeletons lies in their internal operation. MaLLBa::BnB [6]

implements a Branch and Bound technique over the problem search space. It needs some functions to calculate upper and lower bounds of each subproblem, in order to avoid exploring the hole search space. It explores the tree space, branching each subproblem and bounding the worse branches. When the exploration finishes, the solver returns the best solution found. MaLLBa::A* [9] implements general heuristic searches. An A* search is one of the most complex searches that it is able to do. This kind of search is similar to a best-first search, although it implements some more functionalities. This scheme provides to the user the possibility of selecting very different type of searches by using a configuration class. In a simple configuration it can implement a general Branch and Bound technique. Due to the similarities between both skeleton interfaces, we are going to present the problem implementation into MaLLBa::A*, because it is more complex and offers more configuration alternatives. For this problem the required classes have been defined as shown in Figure 4:

- `Problem`. This class stores the characteristics of the problem to solve. `L` and `W` represent the length and width of the material surface. `n` is the number of different patterns. Vectors `l`, `w`, `p` and `x` are defined to represent the length, width, profit and number of available pieces of each pattern. The `bestBound` field allows to have always updated the value of the current best lower bound (best $g(R)$). In the subproblem generation, this make possible to discard subproblems whose upper bound is worse than the current best lower bound. Table `U_2` contains $U_2(x_R, y_R)$ values for the problem instance. The calculation of U_2 is not exactly the one shown in the algorithm. In this class, the user must specify that the problem to solve is a *maximization* problem.

- `SubProblem`. This class represents a node in the tree or search space. It defines the search for a particular problem and it must contain a field of type `Solution` in which store the (partial) solution. For this problem, it represents a *build* or distribution of pieces. The necessary fields are: `g` holds the accumulated profit of the represented build, `h'` stores the remaining estimated profit, the length `l` and width `w` of the build, vector `n` stores the number of used elements of each type of piece and `sol` has the solution represented by the subproblem. The methods to define in this class are: `initSubProblem(pbm, subpbms)` creates one initial subproblem from each different pattern. `lower_bound(pbm)` calculates the subproblem accumulated profit $g(R)$. `upper_bound(pbm, sol)` calculates the estimated total profit $f'(R)$ of the subproblem. `branch(pbm, subpbms)` generates a set of new subproblems from the current subproblem. `branch(pbm, sp, subpbms)` generates a set of new subproblems obtained from the combination of the current subproblem with a given subproblem sp. When creating new subproblems, the current build has to be combined with all previously analysed subproblems. So, we have to implement last `branch` method and indicate to the skeleton that the generation of subproblems is of $dependent$ type. `ifValid(pbm)` decides if a given subproblem has any success expectatives. It is used in the parallel solver to discard subproblems that are not generated in the sequential resolution. `similarTo(sp)` decides if two given subproblems are similar or not. This method is necessary in order to avoid exploring nodes in cases where similar and better nodes have been analysed before. `betterThan(sp)` decides which of two given subproblems is better. It allows to discard the worst of two similar subproblems. Because of the operation of the algorithm, it can be simply implemented with the skeleton.

- `Solution`. This class defines how to represent the solutions. Vectors `x` and `y` represent the position of the bottom left corner of each pattern on the surface and `pattern` contains the sequence of pieces that make up the build.

```
requires class Problem        // Two-Dimensional Cutting Stock Problem
{
   long L;                    // Surface length
   long W;                    // Surface width
   long n;                    // Number of different patterns
   vector<long> l;            // Patterns length
   vector<long> w;            // Patterns width
   vector<long> p;            // Patterns profit
   vector<long> x;            // Number of available elements for each pattern
   Table U2;                  // Dynamic programming table
   Number bestBound;          // Problem best current bound
   ...
}

requires class Solution
{
   vector<long> x;         // X coordinate (position) of the solution patterns
   vector<long> y;         // Y coordinate (position) of the solution patterns
   vector<long> pattern;   // Sequence of patterns that compose the solution
   ...
}

requires class SubProblem
{
   Number g;               // Accumulated profit of the subproblem
   Number h;               // Maximum obtainable profit of the remaining area
   long l;                 // Length of the current build
   long w;                 // Width of the current build
   vector<long> n;         // Number of elements used of each pattern
   Solution sol;           // Solution represented by the subproblem
   ...
```

Figure 4. MaLLBa classes for the 2D Cutting Stock Problem

Once all these specifications for the problem have been done, MaLLBa::A* sequential solver will work in the following way. First, all initial subproblems are created. Initial subproblems are inserted into the open list. For this problem, subproblems must be always inserted into open by order, from higher upper bound to lower. At each iteration, the first subproblem in the search (open) list is removed. If no other similar and better subproblem has been analysed before, it is branched and inserted into the list of best subproblems. All new subproblems or builds are generated by combining (horizontally and vertically) the best current build with all builds previously expanded (builds in best list). The new created subproblems that verify the problem constrains, are inserted into the search list and the value of the best current bound is updated. Last step is repited until the best current subproblem is a total solution, that is, no more profit is obtainable from the remaining material. For its operation, MaLLBa::A* skeleton uses two linked lists: open and best. The open list contains all the nodes generated but not expanded yet and the best list contains the best expanded nodes.

The MaLLBa skeleton provides to the user the following classes:

- `Setup`. This class is used to configurate all the search parameters and skeleton properties. The user can specify if the best list is needed, the type of insertions to do into open, the type of subproblems generation (dependent or independent), if it is necessary to analyse or not similar subproblems, if search over all the space or if stop when the first solution is found, etc.

- `Solver`. Implements the strategy to do: Branch and Bound, Heuristic searches, etc. Usually, each skeleton provides several solvers. Some of them are sequential and other are parallel.

```
 1 void SubProblem::branch(Problem& pbm, vector<SubProblem*>& subpbms) {
 2   vector<SubProblem*> aux_subpbms;
 3   SubProblem *sp, *new_spV, *new_spH;
 4   bool validV, validH;
 5   long size;
 6   ...
 7   // Insert current subproblem into the list of best subproblems
 8   sp = new SubProblem(*this);
 9   pbm.bs.push_back(sp);
10   size = pbm.bs.size();
11   #pragma omp parallel for private(new_spV, new_spH, validV, validH)
12   for (long i = 0; i < size; i++) {  // Generate all new builds
13     new_spV = new SubProblem();
14     new_spH = new SubProblem();
15     validV = validH = true;
16     generateSpbms(*sp, *(pbm.bs[i]), new_spV, new_spH, &validV, &validH, pbm);
17     // Insert only the valid subproblems
18     if (validV) {           // Vertical build
19       new_spV->h = pbm.calculateBound(*new_spV);
20       if (new_spV->g > pbm.bestBound) {
21         #pragma omp critical (bB_update)
22           pbm.bestBound = new_spV->g;
23       }
24       #pragma omp critical (push_sp)
25         aux_subpbms.push_back(new_spV);
26     } else delete (new_spV);
27     if (validH) {           // Horizontal build
28       new_spH->h = pbm.calculateBound(*new_spH);
29       if (new_spH->g > pbm.bestBound) {
30         #pragma omp critical (bB_update)
31           pbm.bestBound = new_spH->g;
32       }
33       #pragma omp critical (push_sp)
34         aux_subpbms.push_back(new_spH);
35   } } else delete (new_spH);
36   // Delete subproblems that will not get to an optimal solution
37   size = aux_subpbms.size();
38   #pragma omp parallel for
39   for (long i = 0; i < size; i++) {
40     if ((aux_subpbms[i]->g + aux_subpbms[i]->h) >= pbm.bestBound) {
41       #pragma omp critical (push_sp)
42         subpbms.push_back(aux_subpbms[i]);
43     } else delete (aux_subpbms[i]);
44 } }
```

Figure 5. Ad hoc parallelization with MaLLBa::BnB

4. Parallel Schemes

The first aproximation followed to solve the presented problem was made into MaLLBa::BnB skeleton. MaLLBa::BnB skeleton provides a structure with the representation of the search space and allows to do a best-first search. But it does not provide any mechanism to store the expanded nodes. So, the difference with the implementation described befored is that in this case, the user must program the way of storing the explored nodes. At each branch, the user has to update the close or best list and do the combination of the current subproblem with all the subproblems expanded before.

Taking into account the necessary computational effort to do the generation of subproblems, it is first proposed the parallelization of the most hard loops in the `branch` method. This parallelization is done by the user using OpenMP (see Figure 5). First, the subproblem generation and verification loop is parallelized (line 11). Each thread does the combination of two builds (horizontal and vertical) and verifies if the new subproblems are valid. The threads must also update the best bound value obtained for the problem. This is a critical operation because `bestBound` is a shared variable (line 21 and line 30). Threads must be also carefully when inserting subproblems into the list of new

subproblems (line 24 and line 33). The loop that deletes the worst subproblems is also parallelized (line 38).

The solver provided by MaLLBa::A* skeleton is based in a shared memory scheme to store the subproblem lists (open and best) used during the search process. Both lists have the same functionality than in the sequential skeleton. The difference is that the lists are now stored in shared memory and it makes possible that several threads can work simultaneously in the generation of subproblems from different nodes. One of the main problems is to hold the open list sorted. The order in the list is necessary to get always the optimal solution. Moreover, if worse subproblems are first branched, the two lists would grow unnecessarily and useless work would be done. By this reason, standard mechanisms for the work distribution could not be used.

The technique applied is based on a *master-slave* model. Before the threads begin to work together, the master generates the initial subproblems. At that moment, the master and the slaves begin their work to solve the problem. At each step, the master extracts the first node of the open list. If it is a solution, the search finishes. In other case, and assuming that the node is inserted in best, next step consists in verifying if there is some one doing its branch or if it had been done before. If the node is still unbranched and nobody is working on it, the master does this work. If the node is assigned, the master must wait until the thread which works on it finishes to generate its subproblems. Once all the node subproblems have been generated, the master inserts them in the open list. Until the master does not notify the end of the search, each slave works generating subproblems from the unexplored nodes of the open list.

In this scheme, some problems appear when different threads are simultaneously trying to modify the same shared variable. By this reason, several mechanisms have been developed in order to guarantee the synchronization of all threads and the consistent view of all shared variables.

5. Computational Results

For the computational study, we have selected some instances from the ones exposed in [7]. The selected problem instances are: $1\_$, $A4$ and $A5$. The experiments have been run on a machine with 4 processors Intel Xeon 1400 MHz and on an Origin 3800. Here the first one are presented.

Table 1 shows execution times and the number of average computed nodes for the executions of the sequential solvers and the parallel solver with 2 and 4 threads in the case of the MaLLBa::BnB and MaLLBa::A* implementations. Times gotten with MaLLBa::A* skeleton are quite lower than the ones obtained with MaLLBa::BnB skeleton. That is because in MaLLBa::BnB implementation the number of computed nodes is higher. In an A* Search the process stops when the first solution is found. But in a Branch and Bound the process must verify, for each branch, if it is necessary to explored it or not. Anyway, the initial parallelization is not very efficient in comparation to its sequential scheme.

	Sequential		2 Threads		4 Threads	
Problem	Comp.	Time	Comp.	Time	Comp.	Time
MaLLBa::BnB						
1_	12979	23,58	13403	30,35	13919	21,61
A4	45033	178,71	45361	211,23	45712	208,91
A5	13668	14,89	13526	31,83	13115	22,43
MaLLBa::A*						
1_	3502	2,78	4136	4,90	4044	3,39
A4	864	75,94	854	9,51	860	7,36
A5	1674	6,17	1510	6,63	1526	3,13

Table 1

6. Conclusions

In this work, we have presented two implementations of the Two-Dimensional Cutting Stock Problem. One implementation is based on a Branch and Bound technique and the other does an A* search. An ad hoc parallelization has been done over the MaLLBa::BnB user code. The MaLLBa::A* skeleton provides a parallel solver that can correctly be used to solve the problem. Both parallelizations have been done with OpenMP.

We have presented computational results obtained for these implementations. As we have shown, the initial parallelization is not efficient because the necessary synchronization needed to updated the shared variables introduces an important overhead. The improvements introduced by the parallel MaLLBa::A* solver are obtained because the number of generated nodes decreases when more threads collaborate in the problem resolution. That is due to the fact that the update of the best current bound is done simultaneous by all threads, allowing to discard subproblems that in the sequential case have to be inserted and explored. Other advantage of this parallel scheme is that the work distribution between the slaves is balanced. Anyway, it is important to consider that the parallelization of an algorithm where the generation of new subproblems depends on all the previous work done, is quite complex.

Actually, we are working to obtain more results with other problems in order to study the behaviour and efficiency of the skeletons in the implementation of different cases. Finally, some work is done in the implementation of a parallel version of the A* scheme based on the message passing paradigm.

References

[1] E. Alba, et al.: MaLLBa: A Library of Skeletons for Combinatorial Optimization. Proceedings of the Euro-Par'02. Springer-Verlag. 2002.

[2] E. Burke and G. Kendall: Applying Simulated Annealing and the No Fit Polygon to the Nesting Problem. Proceedings of the International Conference on Artifi cial Intelligence (IC-AI'99). CSREA Press. 1999.

[3] K. A. Dowsland and W. B. Dowsland: Packing Problems. European Journal of Operational Research. 1992.

[4] H. Dyckhoff: A Typology of Cutting and Packing Problems. European Journal of Operational Research. 1990.

[5] P. C. Gilmore and R. E. Gomory: The Theory and Computation of Knapsack Functions. Operations Research. 1996.

[6] Gonz´alez, J. R., Le´on, C., Rodr´ıguez, C.: An Asynchronous Branch-and-Bound Skeleton for Heterogeneous Clusters. Proceedings of the EuroPVM-MPI'2004. Springer-Verlag. 2004.

[7] M. Hifi : An Improvement of Viswanathan and Bagchi's Exact Algorithm for Constrained Two-Dimensional Cutting Stock. Computer Operations Research. 1997.

[8] S. Maouche and C. Bounsaythip: Optimizing Textile Shape Placement by Tree Genetic Annealing. Proceedings of the Society for Computer Simulation Conference (SCSC'96). 1996.

[9] G. Miranda: Esqueletos Paralelos A*. Aplicaci´on al Problema de Corte Bidimensional. Escuela T´ecnica Superior de Ingenier´ıa Inform´atica. Universidad de La Laguna. 2004.

[10] OpenMP Architecture Review Board: OpenMP C and C++ Application Program Interface. Version 1.0. 1998.

[11] P. E. Sweeney and E. R. Paternoster: Cutting and Packing Problems: A Categorized, Application-Orientated Research Bibliography. Journal of the Operational Research Society. 1992.

[12] K. V. Viswanathan and A. Bagchi: Best-First Search Methods for Constrained Two-Dimensional Cutting Stock Problems. Operations Research. 1993.